Implementing QUIC in Haskell

One of IIJ's goals is to contribute to the development of the Internet, and one way our lab does this is through its involvement in standardizing new protocols. For years, we have been helping to develop more complete specifications. Our work involves discussing new protocol specifications, implementing those specifications, and testing interoperability with other implementations.

Since 2013, I have participated in the standardization of HTTP/2 and TLS 1.3. Over the last two and a half years, I have been involved in the standardization of QUIC and HTTP/3, which are closely related to these two protocols. In this report, I explain how I implemented QUIC and HTTP/3.

3.1 QUIC and HTTP/3

QUIC is a new transport protocol that uses UDP. It is defined as a large specification incorporating the following features.

- Reliability, flow control, and congestion control provided by TCP
- Multiplexing with asynchronous streams derived from HTTP/2 (stream fragmentation and reassembly)
- Security features provided by TLS 1.3 (key exchange, authentication of peers, encryption of data)

The basic units in QUIC are called *packets*. A packet can contain multiple units of data called *frames*. There are several types of frames: e.g., application data is stored in STREAM frames, and ACK (acknowledgement) information is stored in ACK frames. HTTP as defined in the QUIC protocol is called HTTP/3.

3.2 Why Implement it in Haskell?

As with my implementations of HTTP/2 and TLS 1.3, I am implementing QUIC and HTTP/3 in the Haskell programming language. My reasons for choosing Haskell are as follows.

- Rich data types allow for concise problem representation, and strong type checking can detect many coding errors.
- Lightweight threads are provided as standard, enabling threaded programming with better code readability than with event-driven programming and a small overhead when switching and creating threads, where state management tends to be cumbersome (any reference to threads below means lightweight threads).
- Many data types are immutable and can be safely shared between threads.
- STM (Software Transactional Memory) is provided as standard, enabling threaded programming without deadlocks.

Most of the QUIC implementations by other teams use event-driven programming, whereas I use threaded programming. I feel that threaded programming not only improves code readability but also allows me to test specifications from a different perspective than other implementers.

Below, I describe specific implementation points.

3.3 QUIC Streams and Connections

QUIC divides communications into streams in order to multiplex within a single connection. HTTP/2 uses streams for the same purpose, but while HTTP/2 streams can only carry HTTP requests and responses, QUIC streams can carry data for any application.

After working on a QUIC API for quite a while, I discovered the following abstractions.

- The role of QUIC connections corresponds to that of network I/O management handled by the OS.
- QUIC streams correspond to TCP connections.

TCP connections here means the simplest form of TCP connections that only exchange one piece content, as in HTTP/1.0. Viewing things from this angle, I realized that streams can be controlled with an API that mimics the socket API. Part of the current API appears below.

Haskell type annotations are separated by a right arrow. The return type is on the far right. The other parts of the type signature are the argument types. When `IO` appears to the left of the type, it means the method has side effects, such as input and output operations. When `IO` does not appear, the data type is immutable and has no side effects. `()` denotes that there is no return value, and `ByteString` is, of course, the byte string type. So, `IO ()` means that there is no meaningful return value and that only the function's side effects are of interest.

```
-- Abstract data type representing a stream
data Stream

-- Function for creating streams
stream :: Connection -> IO Stream

-- Function for closing streams
closeStream :: Stream -> IO ()

-- Function that accepts streams created by peers
acceptStream :: Connection -> IO Stream

-- Function that receives data from streams
recvStream :: Stream -> Int -> IO ByteString

-- Function for sending data to streams
sendStream :: Stream -> ByteString -> IO ()
```

When implementing an HTTP/1.0 server in Haskell, the usual convention is to use a synchronous approach of starting one thread for each TCP connection from a client, reading a request, writing a response, and then terminating the thread. In HTTP/2, you need to manage multiple threads to enable multiplexing. When implementing HTTP/3, the QUIC library handles this multiplexing. So, when using the above API, it is possible to use the conventional synchronous approach of starting one thread per stream.

3.4 Accepting Connections on a Server

The type annotation of the function that starts a server is as follows.

```
run :: ServerConfig -> (Connection -> IO ()) -> IO ()
```

That is, `run` takes a server configuration and a server application function (a function that receives a connection and does some processing, including input and output) as arguments. The Dispatcher thread launched by this function opens a listening (wildcard) socket for each network interface. When a new connection is accepted, the threads that make up the connection are started (see Section 3.5).

There are six types of QUIC packets. The body of Initial packets, 0-RTT packets, Handshake packets, and 1-RTT packets is encrypted and the header is protected. The body of Version Negotiation packets and Retry packets is not encrypted, nor is the header protected. To analyze these packets in a consistent, unified manner, I devised a method of dividing the analysis into two stages.

(1) Parse parts of the header that are not protected (determine the packet type etc.)
(2) Decrypt encrypted text and remove header protection

Stage (1) is performed by the Dispatcher thread. The Dispatcher thread looks at the results of the analysis in (1) and creates a new connection if it is an Initial packet, or performs the migration process if it is an appropriate 1-RTT packet (see Section 3.9). The specification does not allow the server to accept Version Negotiation packets or Retry packets, so these are simply discarded.

Stage (1) is also performed by the Reader thread described below, and (2) is performed by the Receiver thread described below. The two-stage analysis idea has yielded a common data structure for the header information, resulting in more concise code than in earlier implementations.

3.5 Threads that Make up a Connection

When starting a new connection, the Dispatcher thread starts the main thread for that connection and asks it to create the connection. The main thread starts a group of threads that make up the connection, as shown in Figure 1, and waits for them to finish.

When the connection is created, a connected socket is created. So packets for this connection are read by the Reader thread, not the Dispatcher thread. The Reader performs the packet analysis in (1) above, and passes the parsed header information, protected header, and encrypted body to the Receiver thread through the queue (`RecvQ`).

The Receiver thread performs (2) above, extracts the packet's frames, and processes each of them. STREAM frames are reassembled and passed to the Server thread through the queue (`InputQ`). When it receives an ACK frame, the Receiver thread deletes the corresponding information from the information-retransmission container (`SentPackets`) described in Section 3.10.

The Server thread is what invokes the server application function. The output is sent to the Sender thread through a queue (`OutputQ`). The Server thread is responsible for

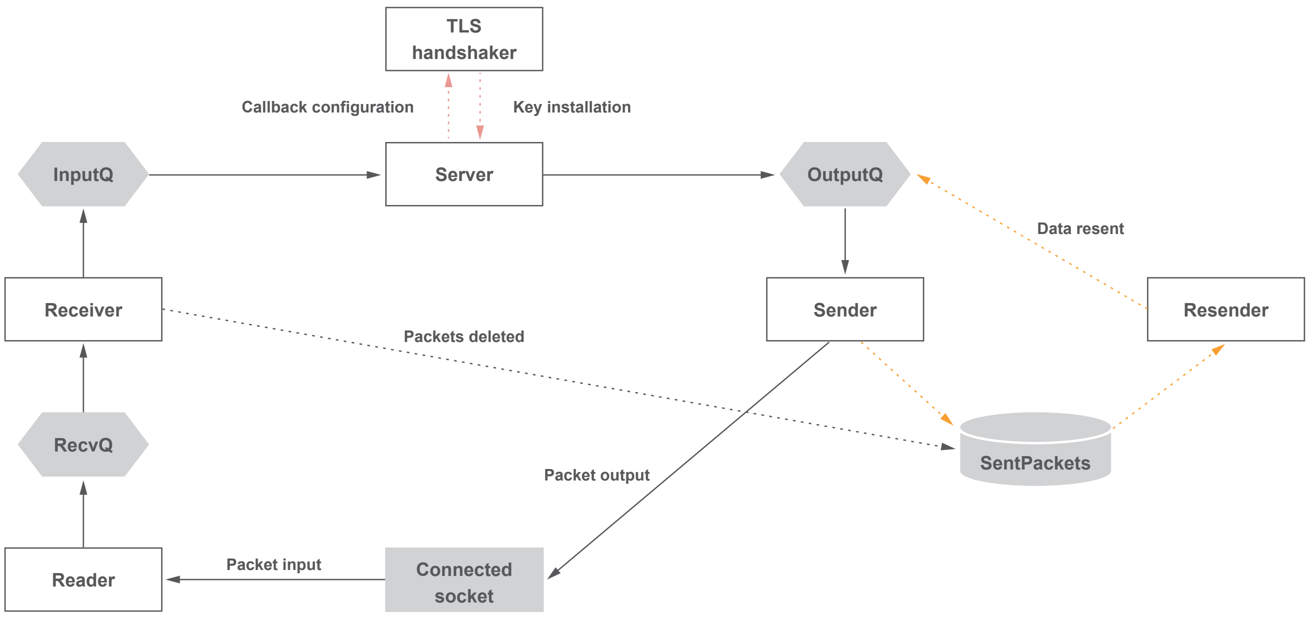

Figure 1: Threads that Make up a Connection

launching the TLS handshaker thread to perform key exchange and synchronize key availability timing before launching the application.

The Server thread is what invokes the server application function. The output is sent to the Sender thread through a queue (`OutputQ`). The Server thread is responsible for launching the TLS handshaker thread to perform key exchange and synchronize key availability timing before launching the application.

When the Resender thread detects a packet loss, it retrieves the relevant information from the information-retransmission container and resends it by putting it into the `OutputQ`.

STM is used for the queues and other data sharing, so these threads do not deadlock. If any one thread causes a fatal error, the entire thread group terminates. When this happens, resources are properly released and no leaks occur.

3.6 Connected Sockets

TCP lets you generate a connected socket from a wildcard socket using the `accept()` system call. The `accept()` system call cannot be used with UDP, however.

For example, suppose your server has a wildcard socket {UDP, 192.0.2.1, 443, *, *} and a client requesting a connection on 203.0.113.1:3456. The connected socket you want to generate is {UDP, 192.0.2.1, 443, 203.0.113.1, 3456}. A simple way to do this is as follows.

(1) Open a new UDP socket and set the `SO_REUSEADDR` option.
(2) Call the `bind()` system call with 192.0.2.1:443.
(3) Call the `connect()` system call with 203.0.113.1:3456.

Unfortunately, on BSD-based OSs, (2) causes an error. Linux allows (2), but race conditions can occur. These problems can be solved as follows.

(1) Open a new UDP socket and set the `SO_REUSEADDR` option.
(2) Call the bind() system call with *:443.
(3) Call the `connect()` system call with 203.0.113.1:3456. In this case, the local address is set to 192.0.2.1.

This method works fine on many operating systems and does not cause race conditions. However, you need to be careful with privileges. Suppose that, in TCP, a process with root privileges creates a wildcard socket for a privileged port. Even if this process relinquishes root privileges for security reasons, the `accept()` system call can still be executed. Linux, however, requires the process to at least have the `CAP_NET_BIND_SERVICE` capability to generate a UDP-connected socket using the above method.

3.7 Closing Connections

When using TCP with the socket API, a call to the `close()` system call by the application immediately returns control to the application, and the OS is then responsible for subsequently terminating TCP. The QUIC implementation also needs to enable this sort of control.

In my implementation, when the server (or client) application function terminates, all threads except the main thread terminate and unnecessary information is discarded. The main thread also starts a separate thread to handle the termination procedure with the minimum information needed to resend the CONNECTION _ CLOSE frame if need be.

In QUIC, an ACK is not returned for packets that contain a CONNECTION _ CLOSE frame. Once the peer has received a CONNECTION _ CLOSE frame, it immediately stops sending packets. So after sending the CONNECTION _ CLOSE frame, we wait a while to make sure that no more packets will arrive from the peer. If packets do arrive, this may indicate that the CONNECTION _ CLOSE frame has been lost, so the packet with the CONNECTION _ CLOSE frame is resent.

3.8 TLS Handshake

QUIC uses TLS 1.3 to perform handshakes to authenticate peers and exchange keys. TLS 1.3 messages are detached from the TLS record layer and stored in a simple data format in CRYPTO frames.

Figure 2 illustrates a full handshake in QUIC.

The client generates Initial keys based on the randomly generated connection ID. The TLS 1.3 ClientHello message is put into a CRYPTO frame, which is then put into the Initial packet, which is encrypted using the Initial key and sent. Note that privacy is not protected because Initial keys can also be generated on intermediate devices.

Upon receiving this, the server generates the Initial key and decrypts the Initial packet. Next, it generates the Handshake key and 1-RTT key based on the retrieved ClientHello. It then puts the generated ServerHello into the Initial packet, encrypts it with the Initial key, and sends it. Other TLS messages are put into Handshake packets and encrypted with the Handshake key before being sent.

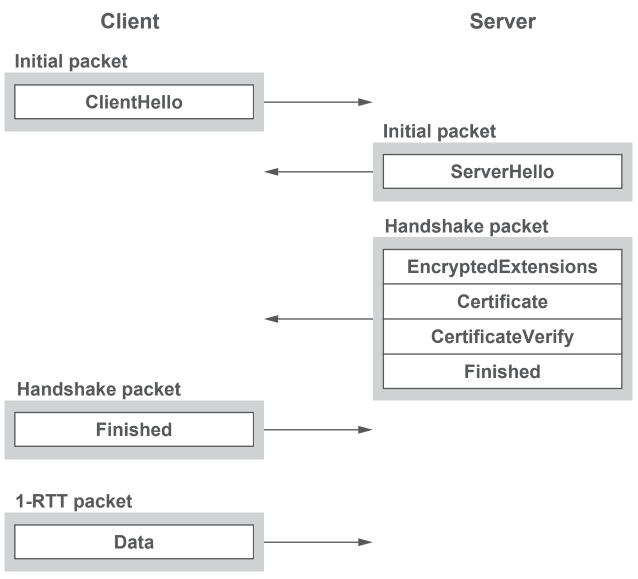

Figure 2: Full QUIC Handshake

A client that receives these packets generates the Handshake key and 1-RTT key. It also puts the generated Finished message into a Handshake packet, encrypts this with the Handshake key, and sends it. At this point, 1-RTT packets capable of storing application data can be sent.

For the second connection, the client can generate a 0-RTT key from the stored information and send an Initial packet followed by a 0-RTT packet capable of storing application data encrypted with the 0-RTT key.

I tried extending the TLS library in various ways to make TLS 1.3 features available in QUIC. Major modifications were needed in order to separate the record layer, but I figured out that starting a dedicated TLS thread was a good way of reusing the TLS library without making any further modifications beyond that.

The callback mechanism proved effective in keeping the state within the scope of the TLS library so that the Client/Server threads do not have to manage the TLS 1.3 state. When a key is generated, a specified callback is used to install the key in the shared data area. And using STM makes it possible for other threads to gauge when the key was installed.

The server-side TLS handshaker thread terminates after sending a NewSessionTicket message in a 1-RTT packet. Meanwhile, the client-side TLS handshaker thread terminates after a set delay upon receiving a HANDSHAKE_DONE frame.

3.9 Migration

Client IP addresses and port numbers can change. This happens, for example, when the network interface switches from mobile phone to Wi-Fi, or when the port mapping on a NAT gateway between the client and server changes. Connection migration is a feature for keeping connections alive in situations like this.

If the client IP address or port number changes, the server side of my implementation will receive 1-RTT packets on the listening socket. By examining the connection ID, it can determine that a migration has occurred rather than a bad packet.

In this case, the Dispatcher thread starts the Migrator thread (Figure 3), which creates a new connected socket, starts a Reader thread that will use that socket, and performs path validation. For details on path validation, see RFC 9000[*1].

Until a new connected socket is created, the Dispatcher thread passes any packets that arrive to the Migrator thread, and the Migrator thread passes them to the Receiver thread. It also closes the old connected socket after a set delay, thereby terminating the old Reader thread.

We will now look at how migration is handled on the client side, starting with the case in which connected sockets are used.

(1) Detect somehow that a new preferred network interface is available.
(2) Call the migration API. Once a new socket is created and the `connect()` system call is called, the OS sets the remote address and port based on the call's arguments. The routing table is then searched using the remote address to find the network interface to which the route points. The IP address of that network interface is chosen as the socket's local address. Local ports are chosen randomly.
(3) Use the connected socket that was created and `send()` to send packets.

The advantage of this method is that path validation can be performed in step (2), and the disadvantage is that OS-specific methods are needed for step (1). Meanwhile, another option is to use wildcard sockets.

- When `sendto()` is called, the OS sets the remote address and port based on the call's arguments. The routing table is also searched using the remote address to find the network interface to which the route points. The IP address of that network interface is chosen as the socket's local address. The local port is chosen randomly when `sendto()` is first called.

The advantage of this method is that migrations happen automatically without the need to keep track of the preferred network interface or provide a special migration API. The disadvantages are the cost and poor performance involved in sending packets and the lack of an opportune time for path validation.

As each method has its advantages and disadvantages, I plan to provide both so that either can be selected via the settings when launching a client.

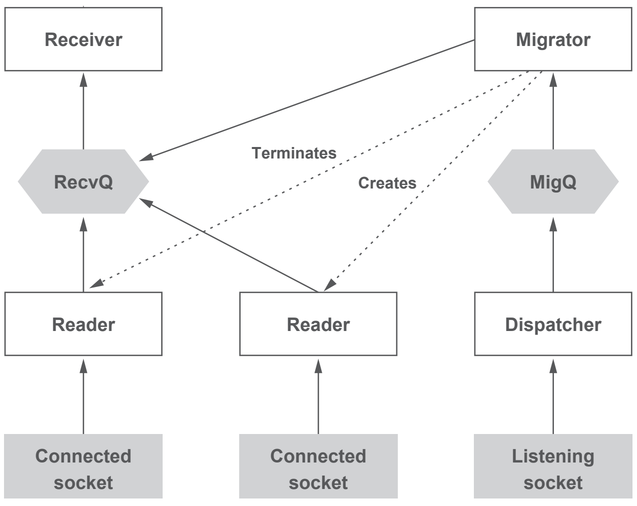

Figure 3: Connection Migration Flow Chart

*1 Reference: J. Iyengar, M. Thomson, "QUIC:A UDP-Based Multiplexed and Secure Transport", RFC 9000, 2021

3.10 ACK Processing Algorithmn

In QUIC, new packet numbers are used for retransmissions. Unlike TCP, which reuses sequence numbers when resending, QUIC has no ACK ambiguity problem. When a packet is resent, the packet number and ciphertext both change. For this reason, RFC 9000 refers to the "retransmission of information" rather than simply to the resending of packets.

In standard TCP, the ACK specifies the sequence number that should be delivered next, so it is not possible to determine whether other TCP segments have been delivered to the peer. QUIC ACK frames, meanwhile, can list packet numbers that have been received.

To enable the retransmission of information carried in packets that have been sent, an information-retransmission container is prepared and the information stored therein at time of transmission. The following three operations can be performed with information-retransmission containers.

- When sending, insert information with the packed number as the key.
- When an ACK is received, delete the information with the packet number as the key.
- If no ACK is returned after a set delay, retrieve and delete the information from the container and retransmit it.

A common data structure in Haskell providing this functionality is the PSQ (Priority Search Queue). We specify the packet number as the key, the transmission time as the priority, and the information as the value.

When I implemented the information-retransmission container with PSQs, I noticed that performance would drop significantly at times. In a normal implementation, for example, say that ACKs are returned as follows.

```
[0,1,2,3]
[4,5,6,7]
[8,9,10,11]
```

That is, the implementation processes ACKs in response to ACKs and dynamically manages which packet numbers need to be ACK'd. At one point, however, Firefox Nightly returned the following ACKs.

```
[0,1,2,3]
[0,1,2,3,4,5,6,7]
[0,1,2,3,4,5,6,7,8,9,10,11]
```

ACKs in response to ACKs are not processed, so unnecessary packet numbers are not deleted. The specification permits this form of ACK. Denoting the size of the PSQ as n and the number of packet numbers specified in the ACK as m, the complexity of the entire delete operation is O(m log n). When m becomes large, as with the Firefox Nightly build I encountered, the delete operation becomes very costly.

I realized that predicates could be used to solve this problem. A list of packet numbers like `[4,5,7,8,9]`, for instance, is represented in an ACK frame in the form of ranges like so: `[(4,5),(7,9)]`. This can be converted into a predicate as follows.

```
predicate :: PlainPacket -> Bool
predicate pkt = (4 <= n && n <= 5) || (7 <= n && n <= 9)
  where
    n = packetNumber pkt
```

Haskell provides as standard a data structure called finger trees (FingerTree), a sequence representation that is easily manipulable at both ends, like a bidirectional list. Finger trees have an operation for splitting themselves into a finger tree that contains only elements matching a predicate and a finger tree holding the non-matching elements, which runs in O(n) time. So using finger trees and predicate-based splitting instead of PSQs, I was able to reduce the computational overhead when ACKs are received.

3.11 Reassembling Streams

QUIC packets do not span multiple IP packets. That is, they are not fragmented or reassembled at the IP level. Data within streams, on the other hand, can span multiple QUIC packets. So the sender needs to split the stream data into appropriately sized fragments, and the receiver needs to reassemble it.

When a STREAM frame arrives, the fragment is inserted into a reassembly container. Then, if there is a continuous set of fragments starting at the expected offset, they are removed and put into the `recvStream` queue. Hence, the reassembly container has insert and retrieve & delete operations.

In the old implementation, I used a one-way list for the reassembly container. Inserts and retrieve & delete operations both ran in O(n) time. When I profiled data transfers in a production environment, I found stream reassembly to be a bottleneck.

This prompted me to adopt a different data structure for the reassembly container: a skew heap populated with finger trees. Elements can be prepended or appended to a finger tree in O(1) time to represent a continuous series of fragments. Computation complexity is reduced: inserts take O(log n) and retrieve & delete operations take O(n) time.

3.12 Flow Control

Flow control is a mechanism whereby senders limit the volume of packets they send to within the bounds of what the receiver can handle. QUIC uses a scheme in which receivers tell senders how much data they can receive (credit). This is often conflated with congestion control, described in Section 3.13, but it is a separate mechanism.

In my implementation, flow control is done at the stream API level.

- `sendStream` sends data within the bounds allowed by the peer, and if the amount exceeds the limit, it waits for credit from the peer.
- `recvStream` assumes that the application will consume this data and sends credit for the amount of data received to the peer.

3.13 Loss Detection and Congestion Control

QUIC loss detection and congestion control are defined in RFC 9002[*2]. Loss detection uses both ACK-based and probe timeout-based methods. And congestion control uses an algorithm based on NewReno. I implemented the pseudocode given in the RFC faithfully in Haskell. In the process of doing so, I discovered, and reported, a number of inconsistencies in the specifications. In recognition of this, the name of this article's author (Kazu Yamamoto) has been added to the RFC 9002 contributors list.

Loss detection and congestion control logs are exported in qlog format[*3] and fed into the qvis[*4] visualization suite to monitor the program's operation and find errors.

*2 Reference: J. Iyengar, I. Swett, "QUIC Loss Detection and Congestion Control", RFC 9002, 2021

*3 Reference: R. Marx, "QUIC and HTTP/3 event definitions for qlog", Internet-Draft, 2020

*4 qvis, "Welcome to qvis v0.1, the QUIC and HTTP/3 visualization toolsuite!" (https://qvis.quictools.info/).

3.14 Testing

My QUIC library and HTTP/3 library implement a variety of unit tests. In this section, I discuss the use of some noteworthy unit tests and external tests.

■ Loss detection

To test if loss detection is working correctly, I implemented a virtual network that relays UDP datagrams through a relay thread. The relay thread drops UDP datagrams based on given scenarios. Naturally, I have implemented tests that randomly drop UDP datagrams. I also comprehensively cover patterns involving handshake packet loss, something that is apt to cause problems, such as tests that drop the client's first packet and tests that drop the second.

■ h3spec

Tests can easily miss error cases. For HTTP/2, `h2spec`[*5] is an excellent test tool for checking if servers can handle error cases. I realized that I could easily test error cases by creating hooks for the Haskell QUIC library. One of the hooks is shown below.

```
onTransportParametersCreated :: Parameters -> Parameters
```

When transport parameters are created, this hook converts one of the parameters from one value to another. An error case can be created by converting to a value that causes an error. Based on this idea, I have released a tool called `h3spec`[*6] for testing error cases against QUIC or HTTP/3 servers. At present, it provides 32 QUIC error tests and 16 HTTP/3 error tests. `h3spec` has been used to test the Haskell QUIC library as well as other implementations, and it has thus played a role in making implementations more stable.

■ QUIC tracker

QUIC tracker is a service that executes a range of tests on public servers once a day and publishes the results. I registered our public server for the service and found a lot of bugs. I was eventually able to pass all test cases except for two unsupported items.

3.15 Acknowledgments

I would like to thank the following individuals. Kazuho Oku gave me the idea for generating connected sockets. Tatsuhiro Tsujikawa discussed the migration API with me. Robin Marx taught me all about qlog and qvis.

Kazu Yamamoto

Head of Development Group, Research Laboratory, IIJ Innovation Institute Inc.
Dr. Yamamoto is interested in applying the concurrent technology of the Haskell programming language to network programming.
He pens the "QUIC wo Yukkuri Kaisetsu" [QUIC at an Easy Pace] series in Japanese on the IIJ Engineers Blog.

*5 h2spec, "A conformance testing tool for HTTP/2 implementation" (https://github.com/summerwind/h2spec).

*6 h3spec, "Test tool for error cases of QUIC and HTTP/3" (https://github.com/kazu-yamamoto/h3spec).